**International Journal of Information and Computer Science**
IJICS Volume 1, Issue 6, September 2012 PP. 135-143
ISSN(online) 2161-5381
ISSN(print) 2161-6450
www.iji-cs.org

# Impacts of Teaching Test-Driven Development to Novice Programmers

Kevin Buffardi, Stephen H. Edwards
Computer Science, Virginia Polytechnic Institute and State University, Blacksburg, Virginia, USA
kbuffardi@vt.edu, edwards@cs.vt.edu

**(Abstract)** Due to the popularity of Test-Driven Development (TDD) among software professionals, some schools have integrated it into their computing curricula. Through exposure to TDD, students gain practical experience while future employers benefit from their familiarity with the technique. However, it is important to investigate empirically whether the use of TDD in the classroom affects student performance or improves the quality of their code. Our study leverages an extensive data set representing multiple years of teaching TDD in introductory computer science classes.

In this paper, we explain methods, tools, and metrics for determining students' adherence to TDD. Our study has the distinct advantage on the analysis of *multiple* snapshots of each student's work throughout the development process, which provides insight into their work habits and consequential outcomes. Furthermore, we establish positive correlations between students' adherence to TDD and both their solution code quality and their thoroughness of testing. As a result, we provide empirical support for the value of TDD in education and identification areas for improving its instruction.

**Keywords:** Test-Driven Development (TDD); Unit Testing; Software Metrics; Automated Grading; Adherence; Computer Science Education.

## 1. INTRODUCTION

Test-Driven Development (TDD) is a software development process that integrates writing software tests with building the solution itself. TDD has two principal concepts: unit testing and testing first. Unit testing emphasizes the writing test cases that independently validate an individual method or similarly small unit. By following unit testing, developers may identify and localize bugs more easily and gain confidence in their solution [1].

More specifically, TDD's test-first approach specifies that writing unit tests ideally should precede writing the solution itself. However, it is important to note that the process is incremental; test-first does not suggest that all test code is completed before beginning any part of the solution. Instead, writing a unit test only precedes writing its corresponding solution unit, before the process is repeated for other units. The iterative nature of TDD is illustrated in **Figure 1**.

Beck popularized TDD in the early 2000's, advocating that it inspires confidence in solutions and also fosters simple designs by concentrating on small, functional modules. Since software maintenance can be a considerable cost, it should be advantageous to develop solutions in small units that are validated with automated unit tests. The potential for developing software with greater confidence and with better design extensibility is appealing.

As a result, several leading software manufacturers have adopted TDD [2], often as part of larger development methods, such as: Agile, Extreme Programming (XP), and Scrum. Studies in industry have also identified benefits to TDD, as discussed in Section 2.1. Likewise, several universities now include TDD in their computer science courses.

Familiarizing students with contemporary techniques is a practical objective in computer science curricula. By practicing such techniques, students may learn to produce

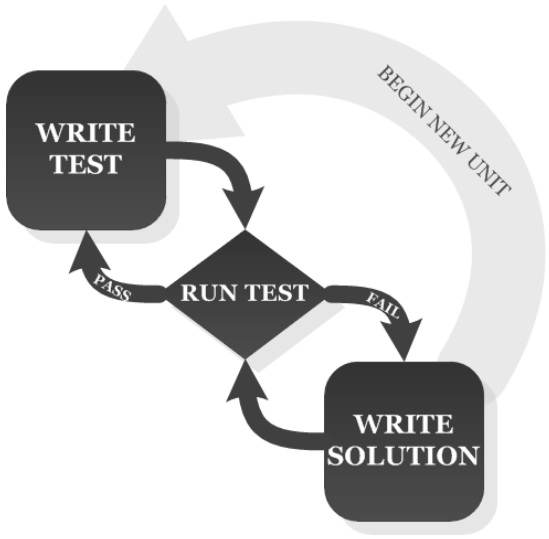


**Figure 1.** Incremental iteration in Test-Driven Development.

higher-quality work. However, the benefits of exercising software development processes should not be limited to improved schoolwork.

Additionally, students should gain experience and confidence in applying the methods they learn in professional settings. The Accreditation Board for Engineering and Technology (ABET) specifically requires that students in computing programs acquire "An ability to use current techniques, skills, and tools necessary for computing practice" [3]. Given TDD's popularity in industry, practicing the technique in computer science classes is valuable.

At Virginia Tech, introductory computer science courses began including TDD in 2003. These courses introduce TDD early and continue to reinforce it throughout the semester by requiring students to write their own software tests for all of their assignment solutions. Despite teaching students how to use TDD and encouraging them to apply its principles in class, we anecdotally observed some students not adhering to its core principles. These observations draw obvious concern.

Supporting and following TDD as supplemental material to the usual introductory courses requires both instructors and students to assume additional workload. To justify the extra burden, it is necessary to investigate the effects of TDD and weigh its merits. In this paper, we assess the impact of using TDD in an academic setting and demonstrate its outcomes. In addition, we identify challenges to improving student adherence to TDD principles.

## 2. BACKGROUND

As discussed in Section 1, Test-Driven Development (TDD) is both practiced by software professionals and taught in computing education. While TDD is popular in industry, it is starting to emerge in computer science curricula. To shape TDD education, we must first understand the advantages and disadvantages in TDD applied in industry. Then, we must identify the unique challenges introduced in an academic context as well.

### 2.1. TDD in Industry

Beck introduced TDD in the context of Extreme Programming (XP) [1]. XP concentrates on values of: communication, simplicity, feedback, courage, and respect [4]. TDD is particularly vital to fostering feedback by systematically conveying the software's current state via unit test results.

However, TDD may be the most difficult aspect of XP to conceptualize and accept [2]. For developers used to a test-after-coding (TAC) approach, testing first is a considerable change in method. Despite this obstacle, a panel of practitioners embraced the incremental test-first approach for its value in producing more verifiable code [2]. Canfora compared TDD to TAC in a controlled experiment and found that TDD improved testing quality [5]. However, along with better tests, they found TDD also slows the development process.

Microsoft also conducted a study reviewing the effectiveness of TDD on multiple software projects [6]. In comparison to non-TDD methods, they reported 15% additional workload upfront to write tests for TDD. Otherwise, their report highly favored TDD. In particular, they observed a considerable improvement in the quality of code produced when teams followed TDD. In addition, they noted extra benefits of unit testing contributing to improved documentation and maintenance.

Meanwhile, Janzen and Saiedian synthesized findings from empirical studies in industry and academia. Their survey of results suggested several common outcomes. In particular, they confirmed that the test-first approach encourages simpler software designs requiring fewer lines of code [7]. They also found a consensus indicating fewer software defects with TDD [8].

### 2.2. TDD in Academia

Early adoption of TDD in a classroom setting has yielded mixed results at different institutions. In an initial study, students generally thought TDD improved the quality of their code [9]. However, instructors found the students were not well-engaged and required additional reinforcement beyond theoretical understanding of TDD. While these findings were preliminary, they introduced both advantages and complications in teaching TDD that resonated in several studies.

Some instructors argued that TDD naturally facilitates active learning by using in-class demonstrations of test validation [10]. In support of active learning techniques, test-first activities may help students develop their skills in analyzing code behavior and predicting how changes to code will alter results [11]. However, to facilitate timely feedback and learn from their analyses, students require an automated assessment tool.

Web-CAT [12] is an automated grading tool that provides students with rapid feedback on their code, including assessments on: validity and completeness of their test code, quality of the solution code, and overall code style [11]. Marmoset [13] similarly assesses students' solutions using instructor-provided tests. Both tools use JUnit [14], the *de facto* standard for writing Java unit tests. ComTest instead uses a proprietary macro language that aims to introduce students with simpler unit tests [15]. However, they also acknowledge a limitation common to the other test-driven learning tools in that test-first is not strictly enforced.

Instructors may consider requiring an early deliverable where students only provide test code. However, it should be pointed out that unit tests in absence of their corresponding solutions are difficult to assess and are not particularly meaningful. In addition, TDD is meant to be incremental so

a large initial set of test cases written before any solution code would violate the principle of incrementally developing (and testing) the product in small units. Consequently, assessing the test-first aspect of TDD is especially difficult.

With the support of test-driven learning tools, a survey of TDD education showed results similar to those in industry: while TDD may increase initial workload, quality of testing and quality of code both improve [16]. In a class using Web-CAT, students produced programs with 45% fewer defects per thousand lines of code when compared to those not using TDD [17]. Nevertheless, academic studies repeatedly find a need to address students' acceptance of TDD.

In particular, younger and less experienced programmers show increased reluctance to adopt TDD [18][19][20]. These findings echo the concern that students require additional motivation to adopt test-first over test-after-coding methods [9]. Likewise, Spacco and Pugh describe a need to use incentives to promote a "test-first mentality" or else students will resort to testing units after writing the solution. Without proper motivation students may avoid testing until after development of their entire solution. Even worse, without requirements to demonstrate comprehensive testing, students may not write any tests at all.

On the contrary, advocating test-first development may not only cultivate practical experience but improve overall learning as well. Writing tests may promote students' critical analysis of their problem solving procedures. In fact, in following TDD, students may replace inadequate "trial-and-error" techniques with productive "reflection-in-action" [21].

Deliberate reflection exemplifies active learning by engaging in Higher Order Thinking Skills (HOTS). Bloom's taxonomy of educational objectives [22] describes how HOTS improve problem-solving skills over Lower Order Thinking Skills (LOTS) such as application of facts and recall of concepts. Instead, HOTS involve more cognitively demanding tasks of analyzing, evaluating, and creating [23].

For example, when a student's unit tests fail, she needs to analyze where her code went wrong and evaluate possibilities for correcting the defect. Consequently, the HOTS required in developing tests and analyzing their results may improve her problem-solving skills. This benefit supplements advantages of gaining practical experience with TDD and producing better solutions.

## 3. METHOD

Current literature on Test-Driven Development (TDD) education is often limited to anecdotal observations and findings from a single semester. Likewise, assessment of students' programming assignments has focused on evaluating their completed code. However, since TDD is a process and not just an outcome, only considering the completed code turned in by a student does not provide adequate insight into the *process* the student followed to get there.

We used Web-CAT as an automated grader for programming assignments. Web-CAT provides rapid feedback to students with assessments of the quality of their solutions, tests, and style. In addition, students are allowed to submit their code to Web-CAT as many times as they like, without penalty—an average of 15 times for each assignment in our study. With each submission, students may review the provided feedback and try to improve their scores. Since each submission is assessed and archived, we gain the unique insight of multiple works-in-progress as students complete their assignments. By considering each submission as a momentary snapshot of the students' development, their history of submissions allows a deeper picture of their activities while developing their solution, and we gain a richer understanding of the processes they follow.

In addition, we have compiled submissions from five years of introductory computer science classes, each including multiple assignments. Every assignment required students to submit test code along with their solution code for assessing test thoroughness. With data from 59678 submissions to Web-CAT, the scale of our analysis is significantly larger than earlier investigations of student testing behaviors. To supplement code analysis, we also surveyed two semesters of students' opinions to dissect and help interpret their behaviors and motivations. In the following sections, we describe our unique method to assessing TDD.

### 3.1. Code Analysis

Over five years (ten academic semesters) of introductory Java courses, we taught TDD and collected students' work on programming assignments. All assignments required unit testing, written in JUnit [14]. As mentioned previously, students submitted their work to Web-CAT for automated feedback and grading. Some assignments included additional, manual evaluation from instructors or teaching assistants. However, only Web-CAT's automated assessment was included to control for individual differences in instructor grading. Scores were normalized for maximum possible score from automatic grading alone.

Web-CAT evaluates students' solutions based on results of instructor-written reference tests. Web-CAT obscures instructor reference tests from the students. Instead, students receive feedback based on the results from the reference tests. Solution *correctness* was calculated based on the percent of instructor's reference tests passed. Web-CAT also uses Clover [24] to evaluate the thoroughness of students' unit tests. Test validity is determined by unit tests passing or failing the solution. Test completeness is assessed by code *coverage*: the percent of solution statements run by the unit tests passed. We concentrated on *correctness* and *coverage* to measure student outcomes in our analysis. **Figure 2** illustrates the evaluation of a student's submission for correctness and coverage.

For the purpose of our discussion, a submission refers to an individual snapshot of a student's work sent by the student to Web-CAT for assessment. For each assignment, each student averaged approximately 15 submissions. Although each submission is assessed, only their final submissions were considered when assigning grades. Accordingly, when describing students' outcomes, we refer to the correctness and coverage of this final submission, unless noted otherwise. To discern students' adherence to TDD principles, we concentrated on the following metrics:

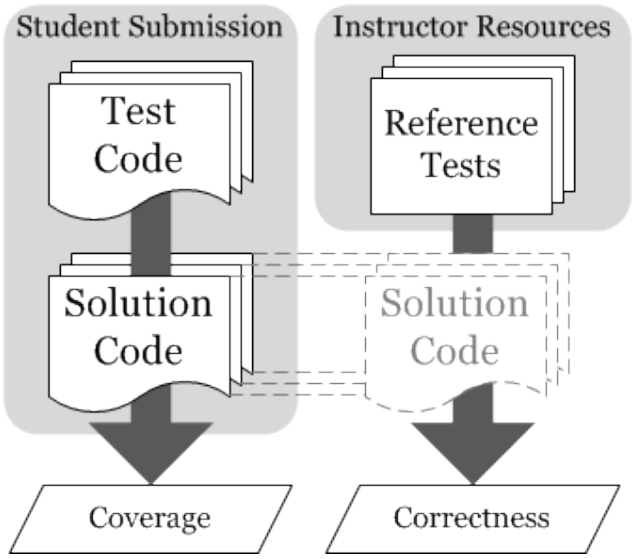


**Figure 2.** Evaluation of outcome metrics based on student code and obscured reference tests

- Test Statements per Solution Statement **(TSSS)**: the number of programming statements in student-written test classes relative to the number of statements in their solution classes.
- Test Methods per Solution Method **(TMSM)**: the number of student-written test methods relative to the number of methods in their solution.

Both TSSS and TMSM measure the amount of test code relative to the amount of solution code. Measuring the absolute amount of code (say, in terms of non-commented source lines of code) would make it difficult to control for the differing size and complexity of various assignments, and also differences between the early or late stages of development of a given student's solution. For instance, in early submissions, having few test statements is not necessarily a concern if the solution also is relatively small. Accordingly, TSSS and TMSM normalize values according to the relative amount of code written by the student so far.

TSSS provides an overall indication of how much work a student has devoted to testing the solution, compared to writing the solution. A low TSSS value (approaching 0) indicates that a student is not dedicating much effort to testing. However, since JUnit test cases typically are less complex than the solution algorithms they test, a TSSS below 1 is expected in most cases.

TDD also emphasizes testing in small units. When incrementally testing and developing units, a JUnit test method corresponds with a solution method that it validates. Accordingly, TMSM provides insight into how well students develop unit tests along with their solution methods.

To observe progress of students' development processes, we recorded these metrics for both the initial and final submissions on each assignment for every student. However, TDD is an incremental process so we also calculated average TSSS and TMSM across all submissions for each student on each assignment. In doing so, we can distinguish students who consistently make progress on developing unit tests from those who may neglect testing for periods only to catch up later.

### 3.2. Survey

Previous research has offered summaries of students' general perspectives of TDD. However, we wanted to dissect those opinions further to understand students' specific impressions of the two key aspects of TDD: testing-first and unit testing. As noted in Section 2, it is particularly difficult to determine whether students wrote tests before their solutions, or simply wrote tests immediately after the corresponding solution. Therefore, we supplemented Web-CAT data with self-reported survey questions on their adherence to TDD. We surveyed students after completing a semester of Software Design and Data Structures (CS2), which follows our introductory course (CS1) and emphasizes TDD. The survey addressed our concerns with the following items:

Rate each item individually on a 5-point scale from 1 (Very Harmful) to 5 (Very Helpful) on the impact the following behaviors have on developing programs:

- Developing thorough test code
- Developing code and corresponding tests in small units at a time
- Developing code and corresponding tests in large portions at a time
- Developing tests before writing solution code
- Developing tests after writing solution code

Students also rated the same items based on the following scale to report students' behavior: Rate each item individually on a 5-point scale from 1 (Very Rarely) to 5 (Very Often) on how often you practice the behavior. Students also reflected on the effect of TDD:

Rate each item individually on a 5-point scale from 1 (Strongly Disagree) to 5 (Strongly Agree) based on your experience with Test-Driven Development (TDD) by incrementally developing tests and then solution code one unit at a time:

- I consistently followed TDD in my programming projects during this course

- TDD helped me write better test code
- TDD helped me write better solution code
- TDD helped me better design my programs
- In the future, I will choose to follow TDD when developing programs outside of this course

The survey was voluntary. After two semesters, we collected 115 responses (49% of the 237 students who took the final exam). Of those responding: 80% were declared computer science majors; 59% previously used Web-CAT before the course; 66% previously had written tests; and 60% had previous experience with TDD.

## 4. RESULTS

### 4.1. Code Analysis

We used the Kolmogorov-Smirnov-Lilliefors test [25] to determine whether each code analysis metric belonged to a statistical normal distribution. **Table 1** summarizes the results, where low p-values support rejecting the null hypothesis that the values are normally distributed.

Initial, final, and average measurements for several metrics were tested independently. As the table shows, each metric's p-value is less than 0.01, which indicates all distributions are non-parametric. Spearman's rho (ρ) [25] determines correlation relationships between non-parametric distributions. Likewise, we used Wilcoxon 2-sample test (p) [25] to compare means since distributions are non-parametric.

To investigate the relationship between testing early in development and final outcomes, we first tested correlations between Test Statements per Solution Statement (TSSS) on a student's very first submission (M=0.68, sd=0.78), and correctness (M=0.79, sd=0.30) and coverage (M=0.87, sd=0.21) achieved on that student's final submission for the same assignment. Results show a nominally positive correlation with both correctness ($\rho = 0.0929$, $p < 0.0001$) and coverage ($\rho = 0.1529$, $p < 0.0001$). The Test Methods per Solution Method (TMSM) of a student's initial submission (M=0.92, sd=0.62) is also positively correlated with correctness ($\rho = 0.2189$, $p < 0.0001$) and coverage ($\rho = 0.1931$, $p < 0.0001$) on the corresponding final submission. In addition, both average (M=0.81, sd=0.67) and final (M=0.88, sd=0.49) TSSS and TMSM (M=1.08, sd=0.58; M=1.20, sd=0.68) values demonstrate increasing strength of correlation with correctness and coverage. All correlations are statistically significant, as shown by low p-values. These correlations are summarized in **Table 2**.

**Table 1.** Mean, standard deviation, and KSL p-values for: correctness, coverage, TSSS, and TMSM.

| Metric | Mean | S.D. | p |
|---|---|---|---|
| Initial | | | |
| TSSS | 0.68 | 0.78 | < 0.01 |
| TMSM | 0.92 | 0.62 | < 0.01 |
| Coverage | 0.61 | 0.37 | < 0.01 |
| Average | | | |
| TSSS | 0.81 | 0.67 | < 0.01 |
| TMSM | 1.09 | 0.58 | < 0.01 |
| Coverage | 0.77 | 0.24 | < 0.01 |
| Final | | | |
| TSSS | 0.88 | 0.49 | < 0.01 |
| TMSM | 1.20 | 0.68 | < 0.01 |
| Coverage | 0.87 | 0.21 | < 0.01 |
| Correctness | 0.79 | 0.30 | < 0.01 |

While all the results are statistically significant ($p < 0.0001$), we considered the possibility that the correlations could be explained by unattributed phenomena. For instance, particularly diligent students may be more likely to follow directions from the instructor. Consequently, they may have produced high quality code regardless of whether they adhered to TDD. However, since they follow directions more closely than less diligent students, a positive correlation for TDD and code quality would emerge.

Likewise, negligent students may write poor code regardless of their adherence to TDD. Since they may not follow directions as closely, they could be less likely to adhere to TDD and would therefore reinforce the positive correlation. Both plausible student stereotypes would strengthen correlation but would *not* suggest that TDD positively affects code quality.

To investigate the possibility of such confounding variables, we categorized students based on performance. Each group was determined based on students' correctness across all of their assignments. Students who achieved at least 80% correctness (which usually corresponds with A and B grades in American schools) on every programming project were designated as "High Achieving." On the other hand, students who consistently achieved less than 80% correctness (usually designated as C, D, and F grades) on every assignment were assigned to the "Low Achieving" group. Lastly, students with at least one assignment under 80% correctness and at least one assignment above 80% were "Mixed Achieving." Approximately 14% of students qualified as Low Achieving, while 54% were Mixed Achieving and 32% were High Achieving.

We controlled for potential effects of particularly motivated and unmotivated students by concentrating on the Mixed Achieving group. This group accounts for the majority of students and each student in the group demonstrated that they have the ability and motivation to write at least one quality programming project, but have also faltered on at least one other programming project as well. Consequently, this group provides an opportunity for investigating the difference in behaviors exhibited by a student when she *succeeds* in comparison to those exhibited when she *fails* another

**Table 2.** Summary of correlations between TSSS, TMSM, and correctness and coverage outcomes.

| Metric | Mean | S.D. | Correctness (M=0.79, sd=0.30) ρ | Correctness p | Coverage (M=0.87, sd=0.21) ρ | Coverage p |
|---|---|---|---|---|---|---|
| Initial | | | | | | |
| TSSS | 0.67 | 0.78 | 0.0925 | < .0001 | 0.1529 | < .0001 |
| TMSM | 0.92 | 0.62 | 0.2189 | < .0001 | 0.1931 | < .0001 |
| Average | | | | | | |
| TSSS | 0.81 | 0.67 | 0.1515 | < .0001 | 0.2762 | < .0001 |
| TMSM | 1.08 | 0.58 | 0.2886 | < .0001 | 0.2690 | < .0001 |
| Final | | | | | | |
| TSSS | 0.88 | 0.49 | 0.2800 | < .0001 | 0.4086 | < .0001 |
| TMSM | 1.20 | 0.68 | 0.3156 | < .0001 | 0.3049 | < .0001 |

assignment. Moreover, a statistically significant relationship between TDD adherence and project correctness would provide compelling evidence of TDD's effects.

To examine within-subject relationships in the Mixed Scoring group, we performed a Mann–Whitney–Wilcoxon test for repeated measures [25] comparing the average TSSS, TMSM, and coverage between assignments with high- (≥80%) and low- (<80%) scoring correctness. Average TSSS on high-scoring assignments (M=0.87, sd=0.71) was significantly greater ($p < 0.0001$) than that of low-scoring assignments (M=0.67, sd=0.66). Average TMSM was also significantly greater ($p < 0.0001$) on high-scoring assignments (M=1.18, sd=0.62) than low-scoring assignments (M=0.93, sd=0.50). Likewise, average coverage was significantly greater ($p < 0.0001$) for high-scoring assignments (M=0.83, sd=0.17) than for low-scoring assignments (M=0.63, sd=0.29).

For due diligence, we examined the trends of these same metrics across all groups. A post-hoc Tukey test [25] identified differences in average TSSS, TMSM, and coverage between each group. The High Achieving group had statistically greater ($p < 0.0001$) average TSSS (M=0.89, sd=0.64) than the Mixed Achieving (M=0.79, sd=0.69) and Low Achieving (M=0.61, sd=0.46) groups. The Mixed group was also significantly greater ($p < 0.0001$) than the Low group. Likewise, the High group's average TMSM (M=1.16, sd=0.59) was significantly greater ($p < 0.001$) than the Mixed group (M=1.08, sd=0.58) and ($p < 0.0001$) the Low group (M=0.79, sd=0.39). The Mixed group was also significantly greater ($p < 0.0001$) than the Low group. Average coverage was also significantly greater ($p < 0.0001$) for High (M=0.83, sd=0.18) than Mixed (M=0.75, sd=0.25) and Low (M=0.59,

**Table 3.** Correlations grouped by high-, mixed-, and low-achieving students.

| Metric | Mean | S.D. | Correctness (M=0.79, sd=0.30) ρ | Correctness p | Coverage (M=0.87, sd=0.21) ρ | Coverage p |
|---|---|---|---|---|---|---|
| Average TSSS | | | | | | |
| High Achieving | 0.89 | 0.64 | −0.1002 | < 0.001 | 0.1457 | < 0.0001 |
| Mixed Achieving | 0.79 | 0.69 | 0.1670 | < 0.0001 | 0.2863 | < 0.0001 |
| Low Achieving | 0.61 | 0.46 | 0.2930 | < 0.0001 | 0.5015 | < 0.0001 |
| Average TMSM | | | | | | |
| High Achieving | 1.16 | 0.59 | 0.2110 | < 0.0001 | 0.1361 | < 0.0001 |
| Mixed Achieving | 1.08 | 0.58 | 0.2680 | < 0.0001 | 0.2796 | < 0.0001 |
| Low Achieving | 0.79 | 0.39 | 0.2135 | < 0.0001 | 0.3712 | < 0.0001 |
| Average Coverage | | | | | | |
| High Achieving | 0.83 | 0.18 | 0.2116 | < 0.0001 | *0.4741*[A] | *< 0.0001* |
| Mixed Achieving | 0.75 | 0.25 | 0.4624 | < 0.0001 | *0.6623*[A] | *< 0.0001* |
| Low Achieving | 0.59 | 0.29 | 0.4243 | < 0.0001 | *0.8305*[A] | *< 0.0001* |

[A] *- Final Coverage data are subsets of Average Coverage calculations, so strong positive correlations are expected*

sd=0.29) groups. Correspondingly, the Mixed group's average coverage was significantly greater than that of the Low group with a p-value below 0.0001.

Spearman rank tests [25] also indicated correlations between average TSSS, TMSM, coverage and the final correctness and coverage within each of the three groups. **Table 3** comprehensively presents the correlations for each metric, separated by group. Again, all correlations are statistically significant with p-values less than 0.001. Positive correlations with correctness and coverage persist in all three groups, with only one exception: average TSSS for High Achievers demonstrates a trivially-small negative correlation with correctness. Since High Achievers already average over 96% correctness, it is difficult to improve much on that score. In addition, high performing students who are just short of achieving 100% correctness may add extraneous test assertions to identify their last remaining defects, which may explain the slightly negative correlation. Otherwise, all three groups demonstrated small- to moderately-positive correlations between each TDD-indicating metric and the final correctness and coverage. Overall, average coverage had the strongest positive correlations with both correctness and coverage.

Anecdotal observations by course instructors suggest that some students completely ignore TDD and only test en masse at the end of their work, after completing a working solution. They may do so only to satisfy the assignment requirements where part of the grade is dependent on Web-CAT's automated testing assessment. To investigate the consequences of this test-last strategy, we identified all situations where the student's initial submission contained no test code at all. We compared the correctness and coverage outcomes of these late-testers to the rest of the students who demonstrated at least some early testing. Early-testers (M=0.81, sd=0.29) achieved significantly better correctness in their final submissions ($p < 0.0001$) than late-testers (M=0.70, sd=0.38). Likewise, the early-testers (M=0.88, sd=0.18) achieved significantly better coverage ($p < 0.05$) than late-testers (M=0.78, sd=0.33).

Lastly, we wanted to identify what early behaviors students demonstrate when they eventually achieve complete (100%) coverage. We compared the initial coverage, TSSS, and TMSM of students who achieved complete coverage to those with incomplete (< 100%) coverage. Complete testers started with higher ($p < 0.0001$) initial coverage (M=0.68, sd=0.39) than incomplete testers (M=0.61, sd=0.35). Complete testers also began with a greater ($p < 0.0001$) initial TMSM (M=1.05, sd=0.73) than those with incomplete final coverage (M=0.89, sd=0.57). While complete initial TSSS (M=0.74, sd=0.82) was greater than that of the incomplete group (M=0.68, sd=0.71), the difference was not statistically significant (p=0.1761).

## 4.2. Survey

Earlier findings using the same survey suggest that while students generally valued TDD, they did not perceive advantages to testing first. Students confirmed their reluctance to writing tests before solutions, as illustrated in **Figure 3**.

According to their adherence ratings, students were significantly ($p < 0.0001$) more likely to write software tests after writing the corresponding solution code (M=4.3, sd=0.71) than to test first (M=2.0, sd=1.0). However, students rated TDD generally helpful in: writing better test code (M=3.5, sd=1.1), writing better solution code (M=3.5, sd=1.2), and designing better programs (M=3.3, sd=1.1). Additionally, they responded positively to using TDD in the future (M=3.5,

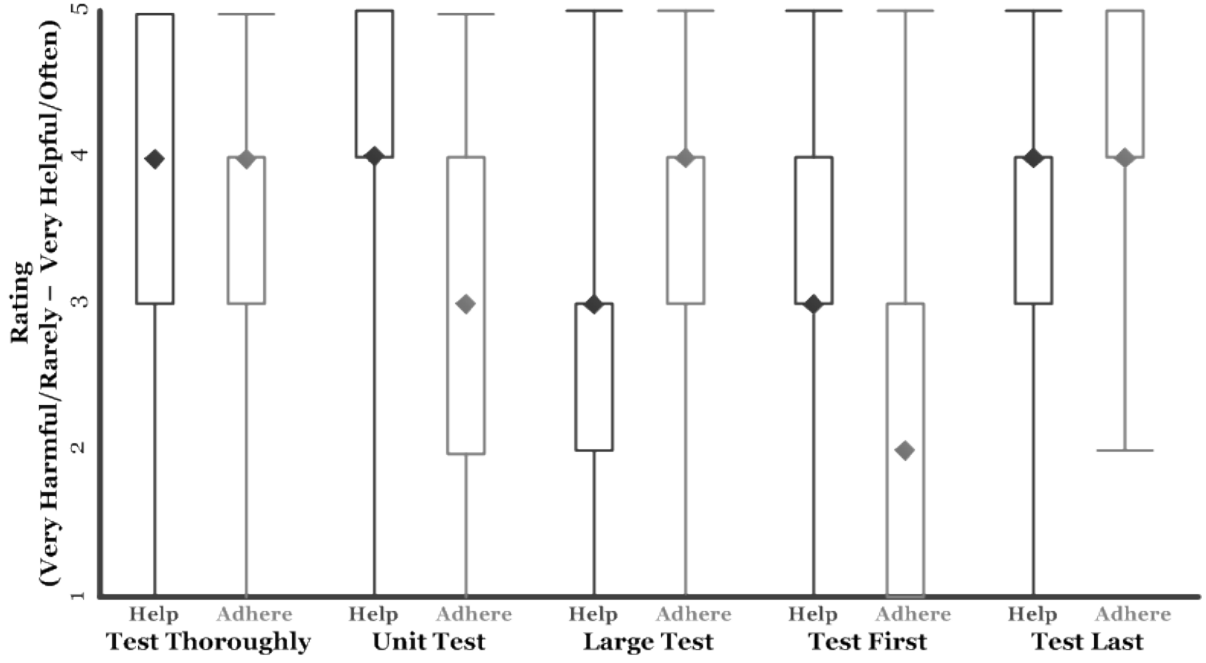


**Figure 3.** Box plot of ratings for helpfulness of- and adherence to- programming behaviors.

sd=1.2).

## 5. DISCUSSION

The measures for Test Statements per Solution Statements (TSSS), Test Methods per Solution Methods (TMSM), and coverage on a student's initial submission all correlated with positive outcomes in terms of final correctness and coverage of the completed solution. This suggests that early testing yields both higher quality tests and solutions. It is also encouraging to find that 87% of first submissions for an assignment include at least some test code. We also see that those who do not test early are less likely to achieve complete coverage by their final submissions, even if they employ extensive testing later in their development. Together, these findings support the claim that Test-Driven Development (TDD) promotes code that is easier to debug and maintain.

TDD also emphasizes testing in small units. We believe that TMSM provides an estimate of how well students comply with unit testing principles by developing corresponding test and solution methods. Meanwhile, TSSS more broadly describes the work put into testing compared to that put into developing a solution. TMSM often had stronger positive correlations with correctness and coverage than TSSS, which suggests that unit testing is particularly beneficial.

However, findings also suggest that success does not only depend on unit testing early, but also on following TDD consistently and incrementally. Higher *average TMSM* over all submissions for an assignment reflects this behavior. Average TMSM shows an even stronger positive correlation with correctness and coverage than TMSM on only the initial submission. It is also a strong endorsement of consistently following TDD since these correlations hold true for all types of students, regardless of their final correctness. That is to say, even if a poorly performing student does not demonstrate the knowledge or effort required to produce a high-quality program, he will still likely improve his solution and testing by following TDD.

While submissions to Web-CAT cannot conclusively indicate whether a student has written software tests first or written solution code first, survey responses confirmed an overall concern with motivating students to adapt a test-first mentality. Overall, students do not consistently practice the principle of testing first. However, as a consolation, they do generally appear to start testing early, even if they "code a little, test a little" [26] instead of "test a little, code a little."

## 6. CONCLUSION

Our study of Test-Driven Development (TDD) featured a previously-unmatched volume of empirical data, spanning ten academic semesters of programming projects. With such extensive insight into student coding practices, we contribute unique findings to the practice and teaching of TDD. Our study presents: novel metrics for assessing adherence to TDD, strong evidence for positive outcomes from following TDD, and direction for improving TDD education.

We developed new metrics for analyzing code that richly describe TDD. Although the measure of tests' coverage was not new, it provides a depiction of the quality or completeness of tests. To complement measurements of test completeness, we focused on adherence to the TDD philosophy of unit testing incrementally. Average Test Methods per Solution Methods (TMSM) in particular helps indicate how consistently a programmer develops unit tests paired with their solution methods.

Previously, evaluating student code was typically restricted to reviewing only the final product of their work. Instead, our analysis leveraged Web-CAT to capture multiple snapshots of students' code while they developed their programs. As a result, we acquired a uniquely rich depiction of the behaviors and processes students follow. With enhanced detail of their processes, project-specific averages of TMSM and coverage revealed how well students demonstrated consistent adherence to TDD over time.

Using these novel metrics on our exceptionally large data set, we add strong evidence that TDD advocates improvements in code and testing quality. We support this validation of TDD with consistent, positive correlations and statistically significant differences in resulting outcomes. Now with improved confidence in the positive impact of following TDD, we can continue advocating and refining TDD education.

However, along with encouraging results of our study, we also confirmed concerns about new programmers' reluctance to adopt TDD's test-first approach. While we found that most students accepted and valued unit testing, we identified testing-first as the most prominent obstacle in TDD adherence. Consequently, it is important for TDD education to investigate interventions and incentives for motivating students to adopt testing-first. Furthermore, future research is required to examine the impact and differences between "code a little, test a little" and "test a little, code a little" approaches.

## REFERENCES

[1] K. Beck,. Embracing change with extreme programming, Computer 32(10): 70-77 (1999)

[2] S. Fraser, S., D. Astels, et al.. Discipline and practices of TDD: (test driven development). Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications. Anaheim, CA, USA (2003)

[3] ABET, Criteria for Accrediting Computing Programs, 2012-2013, Retrieved August, 2012, from http://www.abet.org/computing-criteria-2012-2013

[4] K. Beck, Test-Driven Development by Example, Addison Wesley (2003)

[5] G. Canfora, A. Cimitile, F. Garcia, M. Piattini, and C.A. Visaggio, Evaluating advantages of test driven development: a controlled experiment with professionals. Proceedings of the

2006 ACM/IEEE international symposium on Empirical software engineering. Rio de Janeiro, Brazil (2006)
[6] T. Bhat, and N. Nagappan, Evaluating the efficacy of test-driven development: industrial case studies. Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering. Rio de Janeiro, Brazil (2006)
[7] D. Janzen and H. Saiedian, Does Test-Driven Development Really Improve Software Design Quality?, IEEE Software 25, 2 (2008)
[8] D. Janzen and H. Saiedian, Test-driven development concepts, taxonomy, and future direction, Computer 38, 9 (2005)
[9] E.G. Barriocanal, M.-Á.S. Urban, I.A. Cuevas, P.D. Pérez, An experience in integrating automated unit testing practices in an introductory programming course, SIGCSE Bulletin, 34, 4 (2002)
[10] J. Adams, Test-driven data structures: revitalizing CS2, SIGCSE Bulletin, 41, 1 (2009)
[11] S.H. Edwards, Rethinking computer science education from a test-first perspective, Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications Anaheim, CA, USA (2003)
[12] S.H. Edwards, Web-CAT, retrieved August 2012, from https://web-cat.cs.vt.edu
[13] J. Spacco, Marmoset, retrieved August 2012, from http://marmoset.cs.umd.edu
[14] JUnit, retrieved August 2012, from http://www.junit.org
[15] V. Lappalainen, J. Itkonen, V. Isomöttönen, S. Kollanus, ComTest: a tool to impart TDD and unit testing to introductory level programming, Proceedings of the fifteenth annual conference on Innovation and technology in computer science education. Bilkent, Ankara, Turkey (2010)
[16] C. Desai, D.S. Janzen, J. Clements, Implications of integrating test-driven development into CS1/CS2 curricula, SIGCSE Bulletin, 41, 1 (2009)
[17] S.H. Edwards, J. Snyder, M.A. Perez-Quinones, A. Allevato, D. Kim, B. Tretola, Comparing effective and ineffective behaviors of student programmers, Proc. of ICER workshop, Berkeley, CA, USA (2009)
[18] D.S. Janzen and H. Saiedian, A Leveled Examination of Test-Driven Development Acceptance, Proceedings of the 29th international conference on Software Engineering, IEEE Computer Society (2007)
[19] G. Melnik and F. Maurer, A cross-program investigation of students' perceptions of agile methods, Proceedings of the 27th international conference on Software engineering. St. Louis, MO, USA (2005)
[20] J. Spacco and W. Pugh, Helping students appreciate test-driven development (TDD), Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications. Portland, Oregon, USA (2006)
[21] S.H. Edwards, Using software testing to move students from trial-and-error to reflection-in-action, SIGCSE Bulletin, 36, 1 (2004)
[22] B.S. Bloom, Editor, Taxonomy of Educational Objectives: The Classification of Educational Goals, Handbook I: Cognitive Domain, Longman Group , United Kingdom (1969)
[23] L.W. Anderson, D. R. Krathwohl, et al, A taxonomy for learning and teaching and assessing: A revision of Bloom's taxonomy of educational objectives, Addison Wesley Longman, (2001)
[24] Clover, retrieved August 2012, from http://www.atlassian.com/software/clover
[25] G.W. Corder and D.I. Foreman, Nonparametric Statistics for Non-Statisticians: A Step-by-Step Approach, Wiley (2009)
[26] K. Beck and E. Gamma, Test Infected: Programmers Love Writing Tests. Java Report, 3, 7 (1998)

## Author Introduction

**Kevin Buffardi** is a PhD candidate in Computer Science at Virginia Tech. He has an MS in Human-Computer Interaction from DePaul University and a BS in Computer Science from University of Mary Washington. His research interests include instructional technology, computer science education, intelligent tutoring systems, and human-computer interaction.

**Stephen H. Edwards** received the BS degree in electrical engineering from the Californa Institute of Technology, and the MS and PhD degrees in computer and information science from the Ohio State University. He is currently an associate professor in the Department of Computer Science at Virginia Tech. His research interests include software engineering, reuse, component-based development, automated testing, formal methods, and programming languages.